新人技術者のための

ロジカル・シンキング入門 第3回

冴木 元



システムの記事　ビギナーズ

※本連載記事の第1回と第2回は，本誌2006年6月号，pp.50-59の特集1 第4章「いかにしてバグの原因を突き止めるか」，pp.60-66の特集1 第5章「プログラミングにおける良いデータ構造とは」として掲載されました．

ここでは設計書を作成する際のポイントについて述べる．新人研修などではきちんと設計書を作成してからコーディングを行うように指導されるが，実際の製品開発では設計書の作成が軽視されていたり，きちんと保守されていないケースがある．本稿では設計書に求められる要件について説明する．また，設計書の表現形式として，ブロック図やフローチャートを利用する際の問題についても議論する．（編集部）

Cさんはある携帯音楽プレーヤの開発に携わっています．携帯音楽プレーヤは，最近話題の人気商品であり，各メーカは開発にしのぎを削っています．そんな中，Cさんもここ1年ばかりソフトウェア開発に奮闘してきました．

「仕様が定まらない」，「開発を始めるとハードウェアが動かない」，「機能ブロックを結合すると，システムがハングアップしたり，音が出なくなったりする」など，お決まりのトラブルをくぐり抜けてきたCさんにも，開発がひと段落して，ホッとひと息つく時間ができたようです．

…と思っていたら，上司から以下のような要望が出てきました．「1次製品の開発中のバグは，その数の多さもさることながら，バグが報告されてから解決するまでに時間を費やしたことが問題視されている．2次製品の開発に向けて，各機能ブロックの担当者には，設計レベルからの見直しが必要なのではないか」．

そこでCさんは，各機能ブロックの担当者と連絡をとり，現状，どのような資料が残っているのかを調べてもらうことにしました．そこで判明したのは，携帯音楽プレーヤ全体の仕様書や，各機能を最初に割りふる際に議論した議事録のたぐいは残っているのに，設計書と呼べるものがほとんどないことでした．「設計書？ 仕様書なら見たことあるけどね」と，別段気にもとめないツワモノも数多くいました．一部の機能については，コーディングの前にフローチャートをまとめた人がいるということだったので，それを見せてもらいました．しかし，その中身は現在の構成とはかなり異なるものでした．というのは，途中で加えられたバグの修正や仕様変更がそのフローチャートに反映されていなかったのです．

Cさんの報告を聞いて業を煮やした上司は，次のように断言しました．「2次開発では，設計段階で，各機能ブロックの担当者に詳細な設計書の提出を要求する．それらのレビューを終了してからでないと，コーディングに入ることはまかりならん」．

確かに「設計」の工程はお世辞にも充実していたとは言えず，設計の見直しが必要という上司の指摘は正論であるには違いありません．しかし，問題はそれに必要な工数です．開発対象の携帯音楽プレーヤは，全体で数十万ステップの規模を持っていました．それらを詳細なフローチャートに起こしてレビューし，しかる後にコーディングを行い，発生したバグの修正や仕様変更を正確に反映しようとしたら，どう考えてもコーディングに必要と考えられる期間の2倍程度の工数を上乗せしないと達成できません．2次製品の開発に用意された期間からいって，これは不可能です．

詳細な設計書を用意しろといっても工数的に難しいものがある，とCさんはおそるおそる上司に告げましたが，上司はさらなる正論で追い討ちをかけてきました．「そもそ

KeyWord　設計書，ブロック図，フローチャート，コーディング，ソフトウェア設計，データベース，携帯音楽プレーヤ，DMA，アプリケーション・サーバ，インターフェース

【理想】
- 設計段階で問題点の早期発見に努めること
- コーディングとは，設計書を忠実に移し変える作業にほかならない
- 名人芸に頼ったコーディングからは，安定したソフトウェアは生まれない

【現実】
- ソースが設計書でしょ．やっぱし…
- 納品前に整えれば客は黙らせられるさ
- 最近はソースからドキュメントを起こすツールがあるでしょ

図1　設計書をめぐる理想と現実のへだたり
これは今に始まったことではない．ソフトウェア工学のたいていの本には設計段階における作り込みの重要性が強調されているし，企業の新人研修などでも，「いきなり（キーボードを）打つな」とかならず言われるのだが…

も設計という工程が必要なのは，コーディングする前に問題点を洗い出してしまうことにある．理想のコーディング工程というものは，設計書を機械的に移し変える作業になるはずだ．新人研修でそんなことを言っていたのはおまえ自身じゃなかったか？ おまえにしては珍しくいいことを言うなと思って聞いていたのに，みずから覆してどうする」．

上司の正論を実行に移すために，Cさんに可能な手立てとして，どのようなものがあるのでしょうか（**図1**）．

●「言うはやすし，行うは難し」の設計書作成

Cさんのように，「ソフトウェア設計」の工程のたいせつさがわかっていながら，いざそれを実行に移すとなると，どのように設計書を作ったらいいのかとまどう人は意外に多いようです．確かに新人研修などでは，だれでも設計書を書かされます．そして教師役の先輩社員から，「趣味でプログラムを書くのではないのだから，設計書抜きにいきなりコーディングしないこと」と言われ，フローチャートを詳細に書かされてからコーディングさせられるものです．

しかし，実際に配属されて実務に携わるようになると，「設計書」と呼ばれるドキュメントが用意されることが驚くほど少ないことに気づきます．とくに，Cさんの上司が述べた正論のように，設計書の作成がコーディングの前に行われる，というのはそれほど多くありません．納品の前に，ドキュメント整理のために初めて設計書を起こし，コーディング開始の前に作ったかのように日付を記入して納品した経験のある人が，少なからずいると思われます．

ここで，設計書が作られない理由の典型として，以下のようなものがあります．

- 時間がない

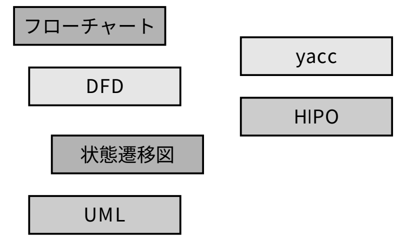


図2　ソフトウェア設計のフォーマット
いにしえから数あるソフトウェア設計のフォーマット．ものの本を調べると，昔からあるフローチャートから最近のUMLまで，いろいろとひな型らしきものがあるにはある．しかし，いかなるシステムにも適応可能な万能のフォーマットは，残念ながらなさそうだ

- 仕様変更が入れば作り直しが必要
- 作ってもだれも見ない

とくに，「時間に制約があり，作っていられない」というのが，理由としてもっとも多いようです．それだけが理由なら，短時間で作成できて，後工程の土台となりうる設計書の表現形式（フォーマット）が決まれば，解決するはずです．ただし，問題はその中身でしょう．

失敗したプロジェクトを調べてみると，「ソフトウェア設計の段階の検討が不十分だったため，後のテスト工程で手戻りが生じた」と言われることがよくあります．そして，実際に設計に携わっていた人に話を聞いてみると，まちがいなく上に挙げたような返答が得られるものです．設計工程の重要さを説く人は多いのですが，「必要な設計書をいかにして作るか」ということについては，答えに窮するケースが少なくありません．

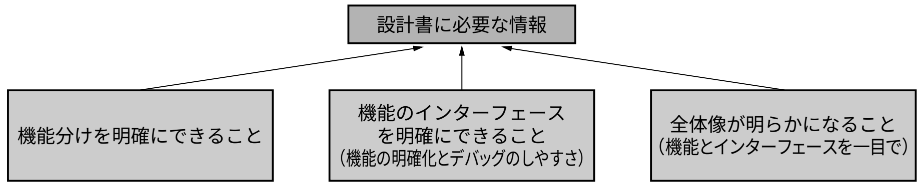
設計書を用意するにあたって，これらの障害があることは無視できません．確かに，短時間に作成できて，後工程の土台となりうるフォーマット（**図2**）があれば，「時間がなくて作れない」といった障害は克服できるでしょう．しかし，設計書を作るうえでいちばん考えなければならないのは，そもそも「どのような情報が設計書に整理されているべきか」ということではないかと筆者は思います．設計書に必要な情報がきちんと整理されていれば，フォーマットはそのときどきで選択できます．逆にフォーマットだけを決めてしまうと，不慣れなシステムを開発するときに足元をすくわれることになりかねません．経験値がマイナスに作用する，ということはどんな場合にも起こりえます．

● 設計書に求められる四つの項目

Cさんたちに作ってもらえそうな設計書を考える前に，まず，「チーム開発において必要な設計書に求められるも

図3
設計書に求められるものとは

設計書に求められるものを挙げてみた．設計書というからには，各ブロックの機能分け，機能間の関係（インターフェース），そしてそれらの全体像を明らかにしたいところ



のは何か？」ということから考えてみましょう（**図3**）．

1）機能分けを明確にする設計書

まず，設計書に必要な情報は，システムの機能分けがいかに行われているかということです．システムの機能分けは，必須機能の整理とともに，以下の作業の基礎にもなります．

- 担当分け
- 作業量や予算の見積もり
- 実装方法の検討

要求仕様を満たすためにどのような機能が必要かを考えることで，必然的に機能分化がなされ，それを基礎に担当分けや作業量の見積もりが行われるからです．実装方法についても，ハードウェアとソフトウェアのどちらを使うかのような大枠のところは，最初に決めなければなりません．

携帯音楽プレーヤのシステム構成がいかに複雑であっても，顧客が求めていることはシンプルで，「いつでもどこでも音楽が手軽に楽しめること」にほかなりません．このことのために，どのような機能ブロックが必要か，というところから設計の作業は始まります．

すべての機能を自分で作るのなら話は別ですが，実際の製品開発では複数の部品を複数の会社に発注して実現することが一般的です．機能分化がなされていないと，開発チームを構成することすらできません．開発をとりまとめる人であれば，機能ごとに何人月の工数が必要かを考える一方で，これまでいっしょに開発してきた各チームのリーダの顔を思い浮かべ，どこの会社のだれに頼めばきちんとやってくれそうかも考えるものです．

例えば，楽曲のデコーダはソフトウェアで実現するのか，それとも専用ハードウェアを組み込むのか，さらにそれは内製（自社開発）するのか，IP（intellectual property）コアを外の会社から購入するのか．楽曲を記憶する媒体はハード・ディスク装置なのか，それともフラッシュ・メモリなのか．こういったことは，開発のかなり初期の段階で決められます．

一つの開発チーム内に限定しても，機能分けの作業の大枠は同じです．どのようにサブ機能を分化したら自分の担当ブロックを実現できるのかを考え，開発するサブブロックの開発規模の見積もりと実際の割りふりを考えます．

以上の説明から，設計書に必要なことは，まず機能分けの明確化であると言えます．

2）機能間のインターフェースを明確にする設計書

機能分けと同時に検討を要するのが，機能と機能の間の関係，つまり，それらの間でやり取りされるデータにどのようなものがあるかです．機能間のインターフェースを決めるうえで考えなければならないことを以下に挙げてみました．

- 各単体機能のデータ・フロー
- デバッグのしやすさ
- 単体テスト項目

機能分けを行ったら，それらの関係がわかりやすくシンプルであることが大事です．複雑に絡み合っているとデバッグの妨げとなり，機能単体のテスト項目も立てづらくなります．

このことを，携帯音楽プレーヤのデータベースとその上位ブロックの関係を例にとって考えてみましょう．例えば，楽曲データを検索するには，最低限，楽曲に対応するキー項目がデータベースに必要です．ユーザが楽曲を選ぶ画面を表示するには，キーから検索された楽曲名などの情報がデータベースからユーザ・インターフェースまで上がってこなければなりません．

そして，ユーザが楽曲を選ぶときに必要な情報は，ユーザがパソコンから楽曲を取り込んで登録したときにはすでにデータベースに登録されていないといけないのです．

このように，ユーザ向けのインターフェースを決めていく過程で，データベースのテーブル仕様はかなり検討が進むはずです．そして，データベースの仕様を明らかにするうえで最初から重要な役割を果たすのは，データベースと上位の機能がやり取りするデータがどのようなものか，ということです．

第1段階で機能分けができたら，それらの間でどのような

データがやり取りされなければならないかを考えて，機能そのものをより詳細に検討していきます．機能分けが済んで機能間のインターフェースが完全に固まれば，工程を一つ進めて，各機能のより詳細な検討に入ることができます．

逆に言うと，機能間のデータに検討もれがあって，それが結合テストの段階で判明すると，たいへん影響が大きくなるのです．例えば，データ登録のときに必要なデータが仕様上で欠けていたという場合を考えてみましょう．この場合は最悪で，パソコンのアプリケーション（ソフトウェア），ユーザ・インターフェース・アプリケーション・サーバ，データベースのすべてに変更が入ってしまうことになります（機能分けの全体像については後述）．

よく言われる「設計段階での検討もれ」というのは，実はそのほとんどが，機能間のデータに検討もれがあった場合なのです．

機能間のデータ・インターフェースを決めるうえでもう一つ重要なことは，デバッグしやすい構成にすることです．システム全体をデバッグするときに重要となるのは，バグの原因となっている機能ブロックの特定です．バグの原因を特定するためには，機能間のインターフェースのデータ・フローをファイルにして取り出したり，ロジック・アナライザでプローブしたりします．

逆に言うと，「デバッグしやすくするには，どういうインターフェースにしておけばよいのだろう」ということを，設計の段階から考えておくべきです．機能間のデータ・インターフェースは，いわばバグを捕らえるための「落とし穴」と言えます（**図4**）．

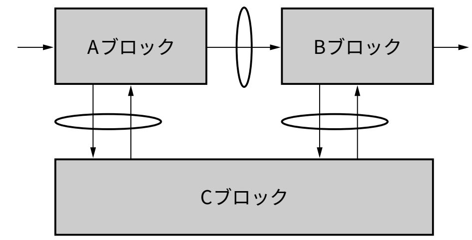


図4　インターフェースはデバッグ時の「落とし穴」を考えて
ソフトウェアを設計する際に，「どう作ればデバッグしやすいか」を考えて設計することが大事．例えば，デバッグのためのビルドを行うと，ログが吐き出され，たいていのバグを検出できるような機能を組み込んでおく

デバッグのための「落とし穴」としての機能間のインターフェースは，テスト工程の側から見ると，取りも直さず各機能ブロックの単体テスト項目の基礎となります．というのは，機能間のインターフェースの動作が確実に保証された段階で結合テストが可能となるため，機能間のインターフェースが設計仕様を満たしているかどうかを確認することが単体テストの目標となるからです．単体機能の確認が完了したら後工程の結合テストを確実に行えるように，設計の段階からくふうされていなければなりません．

したがって，「テストしやすく設計する」のも重要なポイントとなります．デバッグのポイントは，テスト項目の裏返しでもあるのです．

3）全体像を明らかにする設計書

機能分けと機能間のデータが決まれば，設計書を見たときにそのシステムの全体像が明らかになってきます．設計書に書かれた機能とその間のインターフェースから読む人に伝わることは，結局，全体のイメージです．

システム全体の設計書であれば，読む人がその全体像を明確にとらえられるようになっていなければなりません．サブシステムの設計書なら，そのサブシステムの機能分けや機能間のインターフェース，およびほかのサブシステムとのインターフェースが全体像として明らかになっていなければなりません．

ですから「設計書」と呼ばれるものは，人間がぱっと見てわかるように，1枚にまとまって記述されていることが望ましいと言えます．設計書をぱっと見たときに全体構成が明らかで，機能分けと機能間のインターフェースに矛盾が見つからなくなれば，その設計書は完成したと言ってよいでしょう．

● ブロック図でシステムの全体像を記述する

以上のような設計書に対する要求を満たしてくれるフォーマットの例として，ここではハードウェアの外部設計などでも用いられる「ブロック図」を考えてみます．

図5に示したのは，携帯音楽プレーヤ全体のブロック図です．ユーザは，USBケーブルを介して端末（プレーヤ）をパソコンとつなぎ，自分の好きな楽曲データを取り込み，端末内に保存します．そして好きなときにそれらを検索して聴きます．この一連の操作を実現するための各機能とその関係の概略図です．もちろん，このレベルの記述はまだ概略にすぎず，実際に作るとなると，もっといろいろと考えなければならないことが出てくるでしょう．

全体像が決まったら，次に個別のブロックをブレークダウンしてみましょう．例えば，**図5**でさりげなく（？）「アプリケーション・サーバ」と書いた機能ブロックは，実際

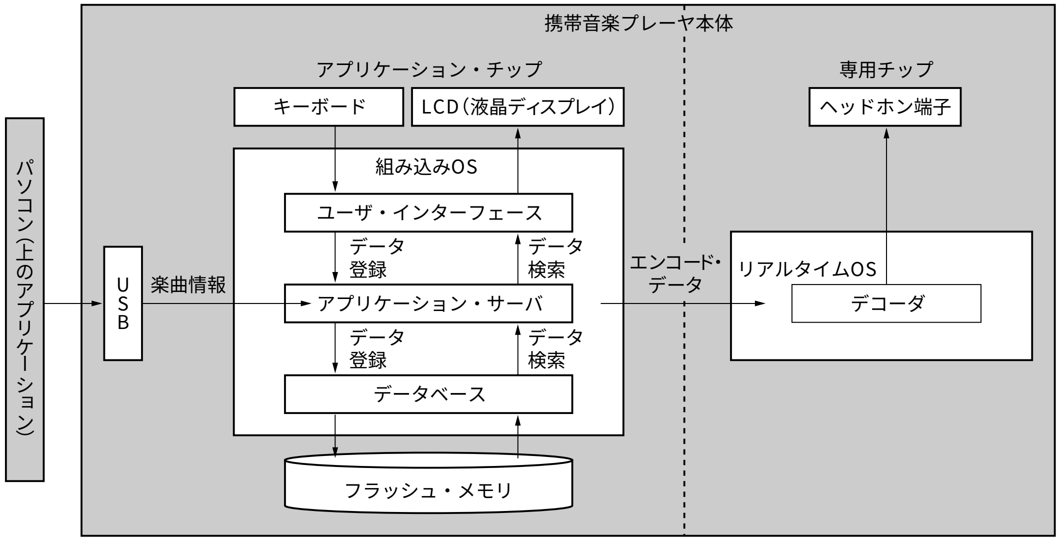


図5
携帯音楽プレーヤの全体ブロック図
おおまかに機能分けを行い，機能間のインターフェースを整理する．まず全体像を明らかにし，次いで，細部のブレークダウンを行う．各機能のすみ分けが明らかになり，機能間のインターフェースが整理されれば，各機能単体の検討に移る

にはいくつかの諸機能からなっているはずです．おおまかに分ければ，登録タスク，検索タスク，演奏タスクの三つの機能が必要です．

では，それらの間のシーケンスはどう記述するのでしょう．例えば，USBケーブルで接続してパソコン側のアプリケーションを立ち上げると，端末側では自動的に登録タスクが立ち上がるようにしたいけれども，そのとき演奏中であったらどうするか，などといった仕様を詰める場合，全体のブロック図だけではとうてい追いつかないでしょう．全体の設計図から一歩進んで，アプリケーション・サーバ内部の仕様を決める設計書を別に起こす必要があります．

ブロック図を作成する際にとくに注意が必要なのは，機能間のインターフェース，すなわち機能ブロック間をつなぐ信号線です．ソフトウェアの設計理論では，よく「モジュールの独立性が高いことが望ましい」と言われます．これは，「モジュール強度が強く，モジュール結合度の弱いモジュールは独立性が高い」という，わかったようなわからないような議論です．

が，この説明も，ブロック図を書いてみれば一目瞭然です．なぜなら，「独立性が強い」ということは，機能間のデータ・インターフェースがすっきりと整理されていて，各機能のすみ分けがはっきりしていることだからです．逆に，検討もれのある設計であれば，ブロックの間の信号線がこんがらがっていて，ブロックの間の分担がわかりにくく，検討の余地が残っていることになります．

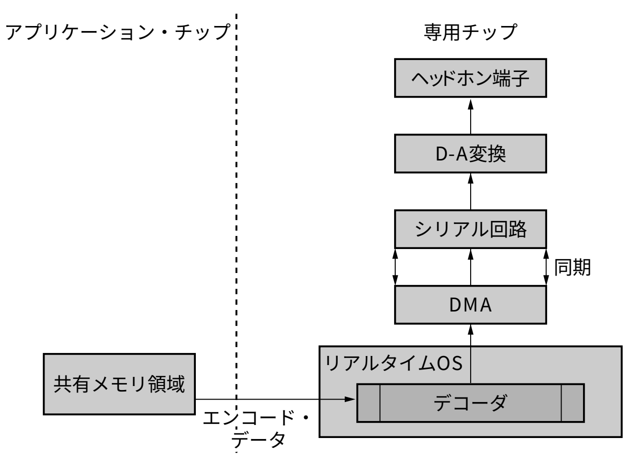


図6　デコーダ機能単体のブロック図
デコーダであるからには，当然，デコード・アルゴリズムを内包しており，この部分の設計とテストはもっとも重要．ただし，周りのハードウェア回路制御を作り込む工数を過少に見積もってはならない

● 機能ブロック単体のブロック図を作成する

各機能ブロックの担当者は，全体ブロックがある程度整理された後で，各機能ブロックの設計に入ります．試しにデコード・ブロックについて，単体のブロック図を描いてみました（**図6**）．

エンコードされたデータは，アプリケーション・チップと専用チップの間の共有メモリ空間に配置することにします．全体のブロック図では記述されていませんでしたが，デコーダでデコードされたデータはシリアル通信回路と同期させたDMA（direct memory access）を使ってD-A変換器につなぎ，ヘッドホンから出力することにします．DMAはデコーダのフレームレートに合わせた周期で割り込みをかけるようにして，その割り込みハンドラからデコ

ーダが呼ばれて楽曲データを処理するしくみにします．

このように整理してくると，デコーダ・ブロックの設計およびテストの工程は，大ざっぱに以下の二つの作業に分かれることが明らかになります．

1）デコーダ・モジュール単体の設計やコーディング，テスト

2）デコーダ・モジュールをリアルタイムに駆動するための周りの部分の設計やコーディング，テスト

もちろん，もっとも工数を要するのは，デコーダ・モジュールそのものの設計やテストであることはまちがいありません．リアルタイム性が厳しく求められるデコード・モジュールを実現するには，最適化設計のノウハウを反映しなければなりませんし，楽曲のデコーダとして満足できる品質を保証するためには，これまた専門性の高いテスト技術が要求されます．

しかしその一方で，デコーダを動かす周りの部分を組み立てる工数を過少に見積もってしまうと，やけどすることになるでしょう．組み込みソフトウェアの開発環境というのは，デバッグそのものよりも，実は環境の立ち上げにかなりの時間が必要です．とくに，いろいろなハードウェアやソフトウェアが複雑に絡むブロックの結合および結合テストは，ソース・コードのステップ数から想像するよりも工数がかかるのが一般的です．

● ブロック図作成の所要時間は比較的少ない

ブロック図を用いた設計書を作成する利点として，そのほかに挙げられるものがあるとしたら，作成にかかる時間が比較的少ないことだと思います．

ブロック図は，全体像を明らかにすることに主眼が置かれています．詳細は省略し，とりあえず全体像1枚を起こせば事足ります．

例えば筆者の場合，設計書がほとんど用意されていないソース・コードを引き継ぐときが時々あります．そのようなときは，とりあえず前任者にお願いして，最低限，ハードウェアとソフトウェアの構成がわかるようなブロック図を機能ブロックごとに作成してもらいます．経験的に言って，それだけであれば，たいていの場合はすぐに用意してもらえます．また，全体像さえ明らかになれば，詳細はソース・コードをたどって何とかやりくりする体制が作れるものです．

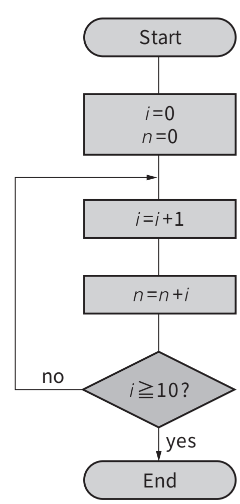
● フローチャートは解決策になるか？

ここまで，ブロック図を用いた設計の概要を説明してきました．次に，「設計書」というとたいていの人が思い浮かべるであろうフローチャートについて考えてみることにしましょう（**図7**）．フローチャートは，冒頭のCさんのような状況に置かれた人に解決策を与えてくれるのでしょうか？

まず試しに，「1から10までの整数を順番に足すプログラム」というものを想定して，フローチャートを作成してみることにしましょう．

フローチャートは**図8**のようなものになるでしょう．「n」に答えの「55」が入るようになっていれば正解です．ここで注意していただきたいのは，フローチャートを使うことによって整理されたプログラム上の情報は何かということです．図を見るとそれは，処理の順序と分岐の条件だけです．

実際，商用プログラムのコーディングでここまで単純なものを書かされることは，まずありません．しかし，どんなに複雑なプログラムであっても，フローチャートが整理してくれる情報の内容が変わるわけではありません．つまり，フローチャートはどのようなプログラムに適用しても，処理の順序と分岐の条件を整理してくれるだけなのです．



▶ 図8
1から10までの足し算のフロー
処理の順序と分岐の条件しか整理できていないのに注目．この程度のアルゴリズムの実装にフローチャートを書き起こすことは実務ではまずないが，整理できる事がらはどんなアルゴリズムに適用しても順序と分岐のみ

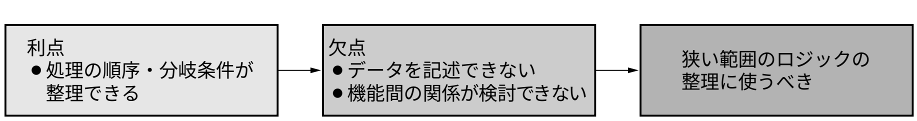


図7　フローチャートの特徴
設計書というとフローチャートを思い浮かべる人は多いが，利点と欠点を見極めて使う必要がある．本稿の結論を述べると，狭い範囲の論理を検討するツールとしての利用にとどめるのが無難．上・中流のソフトウェア設計の工程には向かないと筆者は考えている

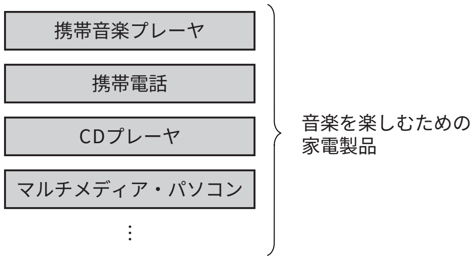


図9　ユーザが製品に求める機能に着目
ユーザが製品に求める機能をユーザの「目的関数」と考えると，各「関数」の作りは製品ごとに異なるだろう．ユーザから見ると似たような「目的関数」でも，作り手から見るとどう違うのかを，設計レベルで検討する必要がある

ですから，**図8**で表せなかった情報は，もっと大きなプログラムの場合にも整理できないでしょう．例えば，ブロック図で重要な役割を果たしていた機能ブロック間のインターフェースの整理に，フローチャートは向きません．フローチャートは，機能ブロックの間のデータの流れを整理することができないからです．

必然的にフローチャートは，狭い範囲の処理を整理するときにのみ使われることになります．複数の機能ブロックの間の関係を整理して，機能分けとその間の信号線を整理するという上・中流のソフトウェア設計の工程ではあまり使われません．

たいていの人にとっては，フローチャートは新人研修のときに書かされたきりで，実際の開発で目にすることは少ないと思います．その理由は，フローチャートの利点を生かせる機会があまりないからだと筆者は思います．

● フローチャート作成にはC記述と同等の時間がかかる

また，フローチャートには，記述するのにとても時間がかかるという欠点もあります．

プロフェッショナルとしてしごとをしているたいていのプログラマなら，**図8**のフローチャートを描くのと同じくらいの時間で，「1から10までの整数を順番に足すプログラム」をC言語などで記述できることでしょう．ということは，例えば1,000ステップのプログラムのフローチャートを描くには，それを書いたのと同じくらいの時間を要すると考えたほうがよさそうです．そこまでしたところで，フローチャートが整理してくれるのは，処理の順序と分岐だけです．商用開発で作られるような規模のプログラムの全ソース・コードに対して，「詳細なフローチャートを書いてからプログラムを起こす」というのはかなり無理があると思います．

● ブロック図による表現が向くものと向かないものがある

この記事を読んで，「自分が開発しているシステムでも設計にブロック図を使うといいかもしれない」と思った方はいますか？ そのようなありがたい方には，自分が実際に開発しているシステムと似た構成のシステムを雑誌の記事などで探して，ブロック図を起こしてみていただきたいと思います．例えば，ゲーム専用機を作っている人なら，それをもし携帯電話に乗せるとしたら，どのような構成になるのか，既存の携帯電話に何を新たに追加して，何をあきらめると性能のいい“ゲーム・ケータイ”になるのかを考えるなどしてみましょう（**図9**）．

そして，具体的にその全体構成を考えたら，自分の担当する機能ブロックの詳細を掘り下げてみます．そうした後で，実際に出荷されている製品の構成と比べれば，自分の見通しがまちがっていた，あるいは古すぎたといったことが判断できたり，新しい製品開発のきっかけが見いだせるかもしれません．そうなれば，ブロック図をあなたが手がけている実際の開発で使ってみるのもいいでしょう．

逆に，ブロック図を作成してみたところ，ふだん自分が使っている設計手法を補うようなものは得られず，肝心の情報が欠落してしまうような印象を持つことがあるかもしれません．この場合は，あなたの開発しているシステムは，ブロック図を設計書に採用するのには向かないので，導入はあきらめたほうがいいかもしれません．

システム開発のいいところは，くふうをすればいろいろなことが自分の手の届く範囲で手軽に試せることであると筆者は考えています．大規模システムの開発が相手でも，くふうしだいで開発手法を思考実験や比較的簡単なテスト・プログラムのレベルに落とし込んで試してみることができます．このように，まず小さめの開発サイクルを回してみる，というアプローチは，ソフトウェア開発においては，けっこう使いでのあるやりかたなのです．

さえき・はじめ

<筆者プロフィール>
冴木元．システム・エンジニア．昔から実は語学マニア．今までマスタしそこねた言語は数知れず．プログラム言語ならなんとかなるのだが…．大学時代は英仏独の文献が読めるようになればいいなと思っていたが，この調子だとライフワークになるかもしれない．